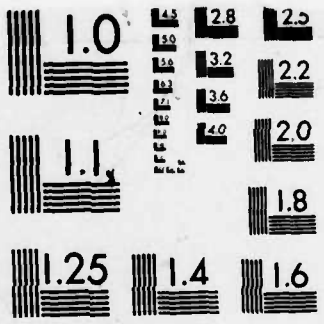
AD-A142 224 AN INFORMAL STUDY OF PROGRAM COMPREHENSION(U) ADVANCED 1/3
INFORMATION AND DECISION SYSTEMS MOUNTAIN VIEW CA
E A DOMESHEK ET AL. MAR 84 AI/DS-TM-1014-3
UNCLASSIFIED AFOSR-TR-84-0309 F49620-81-C-0067 F/G 12/1 NL

END
DATE FILMED
7-84
DTIC

CONT



AI&DS

AD-A142 224

AN INFORMAL STUDY OF PROGRAM COMPREHENSION

Eric A. Domeshek
Daniel G. Shapiro
Jeffrey S. Dean
Brian P. McCune

March 1984

Annual Technical Report for 1 June 1982 - 31 May 1983

Approved for public release; distribution unlimited

Prepared for

Directorate of Mathematical & Information Sciences
Air Force Office of Scientific Research
Bolling Air Force Base, D.C. 20332

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of AI&DS, AFOSR or the US Government.

ADVANCED INFORMATION & DECISION SYSTEMS
Mountain View, CA 94040

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): TM-1014-3	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 84-0309
6a. NAME OF PERFORMING ORGANIZATION: Advanced Information & Decision Systems; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: AFOSR/NM
6c. ADDRESS (City, State and ZIP Code): 201 San Antonio Circle, Suite 286, Mountain View, CA 94040	7b. ADDRESS (City, State and ZIP Code): Bolling AFB, DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: USAF, AFSC, Air Force Office of Scientific Research; 8b. OFFICE SYMBOL (If applicable): FQ8671	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F49260-81-C-0067
8c. ADDRESS (City, State and ZIP Code): Building 410, Bolling AFB, D.C. 20332	10. SOURCE OF FUNDING NOS. — PROGRAM ELEMENT NO: 61102F; PROJECT NO: 2304; TASK NO: A7; WORK UNIT NO:
11. TITLE (Include Security Classification): An Informal Study of Program Comprehension	
12. PERSONAL AUTHOR(S): Domeshek, Eric A.; Shapiro, Daniel G.; Dean, Jeffrey S.; McCune, Brian P.	
13a. TYPE OF REPORT: 2nd Annual Technical Report; 13b. TIME COVERED: FROM 1 June 82 TO 31 May 83; 14. DATE OF REPORT (Yr., Mo., Day): March 1984; 15. PAGE COUNT: 62	
16. SUPPLEMENTARY NOTATION	
17. COSATI CODES — FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number): Program Reference Language (PRL), Extended Program Model (EPM), Intelligent Program Editor (IPE), program documentation, artificial intelligence (AI), knowledge base,

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes work performed during the second year of research on a Program Reference Language. During this year, a study was conducted in which protocols of programmers studying a new program (with the intent of debugging it) were analyzed, both for the vocabulary used and for indications of strategies adopted in their efforts at program comprehension. A sampling of programmers' natural vocabulary for referencing programs was gathered and analyzed. Preliminary steps were taken towards using this data as the basis for the design of a formal query language for the PRL. The study also raised some new issues bearing on the implementation of systems which use the PRL: individual differences imply the need for customization; context-sensitive information management is important; and useful user interface features were identified.

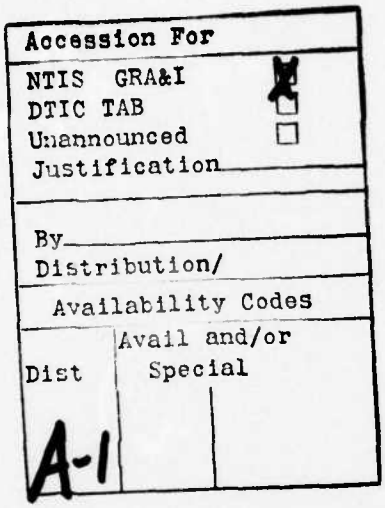
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED [X] SAME AS RPT. [X] DTIC USERS [ ]	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Robert N. Buchal	22b. TELEPHONE NUMBER (Include Area Code): (202) 767-4939; 22c. OFFICE SYMBOL:

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE

18. SUBJECT TERMS (continued)

multiple representations, protocol analysis, user modelling, retrieval language, debugging, program cliches, program annotations.



CONTENTS

CONTENTS

Page

1. INTRODUCTION 1
   1.1 PROJECT HISTORY AND ACCOMPLISHMENTS 1
   1.2 RESEARCH OBJECTIVES 4
   1.3 GUIDE TO READING 5
2. A STUDY OF PROGRAM COMPREHENSION 7
3. DESIGN OF THE STUDY 9
4. THE STANDARD FORM OF THE STUDY 14
5. DATA FROM THE STUDY 18
6. ANALYSIS OF RESULTS 20
   6.1 VARIATIONS IN PROGRAMMERS' STYLES OF PROGRAM UNDERSTANDING 20
   6.2 VOCABULARY FOR REFERRING TO PROGRAMS 22
   6.3 ANALYSIS OF PROGRAM LISTING ANNOTATIONS 24
   6.4 DESIRED DOCUMENTATION 25
   6.5 DESIGN OF THE STUDY 26
7. IMPLICATIONS FOR THE PRL/EPM 28
8. SUMMARY 31
9. PLANS FOR FURTHER DEVELOPMENT 32
10. PERSONNEL 34
   10.1 PERSONNEL 34
   10.2 INTERACTIONS 37
   10.3 PUBLICATIONS 40
11. REFERENCES 44

APPENDIX A. LISP PROGRAM USED IN PRL STUDY 45

APPENDIX B. QUESTIONNAIRE USED FOR PRL STUDY 54

ILLUSTRATIONS

ILLUSTRATIONS

Page

3-1: Documentation Packet fro Program Comprehension Study..... 12

4-1: Instructions for Program Comprehension Study..... 16

1. INTRODUCTION

This report documents the second year of work on the Program Reference Language project (PRL), which is a basic research effort aimed at the creation of a mechanism for flexibly identifying the interesting portions of programs. Although this work began as an investigation into query languages which provided textual and syntactic search predicates, it has grown over the course of our research into a knowledge base about programs in general, and into a database that documents the structures present in specific pieces of code. This development is discussed at length in the annual report for the first year of research, and is recapped only briefly below (See "Searching a Knowledge Base of Programs and Documentation", [Shapiro-83] for more details.) This document focuses on a study of program comprehension which we performed in order to elicit the information required to design a formal query language for the PRL.

1.1 PROJECT HISTORY AND ACCOMPLISHMENTS

Our original proposal defined the Program Reference Language as a tool for flexibly accessing the interesting portions of programs. The project was a two year effort whose goal was to perform research aimed at supporting program creation and mainterance by allowing programmers to locate specific sections of code via textual, syntactic, and historic

search requests (related to a stack of program locations that had been visited).

The PRL was originally intended as a focused tool for accomplishing this end. However, in the course of our research, we developed the hypothesis that the correct approach to supporting program search was to create a system that captured knowledge about the structure of code. At the simplest level, this meant being able to locate loops, procedures, and variables via cross-indexing schemes. On a more sophisticated plane, it meant creating an automated understanding of the structure of programs so that regions of code could be located based on their descriptions. This, in turn, required the definition and support of a vocabulary for referencing code which was in tune with the terminology that programmers natively employed.

We took the task of defining such a vocabulary as the major goal of the PRL. Over the course of our first year of research, it led to the design of a knowledge base that captured much of the syntactic, semantic and pragmatic (domain of application) structures in programs, as well as to the creation of a search mechanism which was able to access that data. During this work, we came to the conclusion that the knowledge required to support program search was in fact identical to the information required to support a variety of intelligent tools for manipulating code. Hence, the PRL grew into a knowledge representation system for recording facts about programs in

general, and into a database for documenting the structures present in specific pieces of code. To emphasize this fact, we renamed the entire system the Extended Program Model (EPM), and consider the PRL to be one part.

At the end of our first year of research, we presented a concept feasibility demonstration of a system which supported program search through a knowledge base representing a toy ADA program. Although it required some admittedly tedious commands, the system was able to answer the request, "find the initializations of the loop which computes the sum of the test scores". This task involved integrating textual and syntactic clues, as well as references to data flow information and programming cliches. The demonstration system is discussed in [Shapiro-83], and (together with the design of the EPM) is the subject of two published papers, at the 1983 Trends and Applications Conference [Shapiro-83b], and the Seventh International Conference on Software Engineering [Shapiro-84].

To provide further context for our work, we also explored an application of the EPM to the program creation process. We outlined a system, called the Intelligent Program Editor (IPE), that employed the EPM's knowledge base to augment the capabilities of standard text editors. Using this information, we felt that the IPE would be able to express semantically oriented consistency constraints, perform large-scale editing transformations, and even provide support for the

template-based creation of programs using the vocabulary defined by the EPM. In January of 1982, the Intelligent Program Editor moved into separate development at AI&DS through a research grant from the Office of Naval Research.

Our plans for the third year of research have focused on transforming the EPM (as defined above) from the concept feasibility phase into a more practical piece of technology. In light of that goal, we undertook the study reported on in this document, which was, in essence, an informal experiment aimed at identifying the vocabulary and the procedures programmers use to search through code. We administered this test to several professional programmers and research personnel within AI&DS. At the time of this writing, the results of the study are being used to motivate the definition (in terms of vocabulary and syntax) of the formal query language which will become the PRL.

1.2 RESEARCH OBJECTIVES

Some of the key research issues which have been addressed in this project are:

1. What are the most useful ways of referring to parts of a program? Said in a different way, what vocabulary do programmers currently use to describe portions of their programs?
2. What information must be included in a knowledge base about pro-

grams and documentation in order for it to support program search?

3. What information must be included in such a knowledge base for it to support a variety of intelligent tools for accessing and manipulating code?

4. How should information of this kind be represented?

5. How should application specific knowledge be included?

6. How can user-supplied assertions and other documentation be acquired and integrated into a knowledge base for use in program referencing and other tasks?

7. How can search requests be expressed in a uniform reference language?

8. What form of a search mechanism is required to implement these reference requests?

9. How can these searches be performed efficiently? In what ways can search be limited or deferred in order to maintain good reponse time?

1.3 GUIDE TO READING

The following chapters provide detail on the study of program comprehension which was performed. Chapter 2 introduces the specific goals of the study, chapter 3 discusses the design of the informal experiment that was conducted, chapter 4 gives the actual information presented to the experimental subjects, chapter 5 describes the data that was collected, chapter 6 begins the analysis of results, and chapter 7 describes the implications of those results for the PRL and EPM Chapter 8 provides a brief summary of this material, while chapter 9 provides a discussion of our future research plans. Chapter 10 contains a discussion of key research

personnel involved with this project.

Appendix A contains the listing of the program used for the study, and Appendix B is the questionnaire that was employed.

2. A STUDY OF PROGRAM COMPREHENSION

To achieve its purpose and to be accepted as a useful tool, the PRL design would do well to pay attention to the way programmers currently perform the common task of learning about an unfamiliar program. Except where it can dramatically increase performance without introducing a prohibitive learning cost, the PRL should present them with a conceptual model of the program that is consistent with the one they use now. It should let them perform operations analogous to the ones they use mentally, but it should speed up the process by keeping all the information integrated together and on-line. It should increase reliability by automatically deriving information whenever possible, to avoid discrepancies between the program text and the support information. Additional functionality may be provided, but it still should aim to stay within programmers' mental model of programs.

Accordingly, we performed a study intended to explore programmers' mental models of programs and their methods for program comprehension. In particular, the focus was on the steps taken in studying an unfamiliar program, and the vocabulary used to describe parts of it. In addition, recommendations were solicited for useful extensions to the capabilities of current support tools. We wanted to know not only what programmers currently do, but also what they wish or imagine they could do if given the right tools to help. This was not meant to be a controlled

experiment; rather, it was an exploratory attempt to gather qualitative data on programmer performance and preferences.

The PRL design incorporates different views of the program, each of which makes different information readily apparent. Some proposed views are similar to the stages of analysis required by compilers to translate a program text into a running program. Some views are also similar to the information captured in certain popular types of documentation, such as flow charts and cross reference listings. Because these types of information are already well known programming tools, it seemed clear that they ought to help program comprehension still more if maintained in an on-line representation that promotes both human and machine processing. One result of this study was to confirm that programmers are already used to thinking about programs in these ways, and have a natural vocabulary for describing parts of programs from these views.

3. DESIGN OF THE STUDY

The goal in this study was to observe programmers going about the business of understanding programs. Rather than hand the subjects a listing and ask them to read it, a debugging task was chosen. Asking the subjects to fix a problem in some piece of code gave them a common task to focus on, and stronger motivation to learn about the program. It was also hoped that the time needed to complete the task would provide some measure of how well they were comprehending the program.

It was desired that the program used for this study reflect as much as possible the realities of code as it is found in the normal production environment. There were of course limits on the size of the program that subjects could be expected to study. A Lisp program which had originally been written for an earlier study of debugging [Shapiro-81] was chosen as the sample program. It seemed a good choice because it contained a single bug that was somewhat subtle, but still deemed discoverable. The program is about 300 lines long, and its listing is approximately seven pages; it is included as Appendix A. The important features of this program, aside from its language and length, are that it was new to all of the subjects in the study, and that it was presented in a fairly scrambled state. This disorganization was the result of porting the program across three different environments and was of value in the study because it truly reflects the state of many

programs.

Programmers generally do not perform their job in a vacuum; they have some sort of support environment. That support is normally composed both of software tools and documentation. However, for the purposes of the study, it was decided to eliminate the tools so that the programmers' work on the program, rather than their facility with certain existing tools, could be observed. Thus even though the program was in Lisp, the programmers did not have access to a Lisp interpreter or environment.

It was also decided to pare back the documentation to the bare minimum, once again to reflect common real world conditions. As an aid to designing this study, a single subject was run in a pilot trial. The pilot subject's ignorance turned out to be far too extensive. He was given the program listing and told to find the bug. He was given no documentation at first: no information about the purpose of the program, its inputs and outputs, or even the external manifestation of the bug. After an hour of studying the listing it became clear that more information was needed. It was necessary to ask the program's author for a brief sketch of the program's purpose and design, and its incorrect I/O behavior.

Based on this experience, it was decided to provide a limited amount of documentation. A short packet was prepared, describing the

purpose and workings of the program followed by a bug report with pictures of several successive output states illustrating the bug. This information packet is shown in Figure 3-1. The subjects were also allowed to refer to the language manual for the dialect of Lisp used. The program itself had only a minimal number of comments written by the original author for his own use. No information on the history of the program was provided.

One interesting observation in the pilot study was the importance of writing on the listing. The subject marked up the listing with highlighters, pens, and pencils in multiple colors. Marginal notes and assorted doodles left a trail indicating which parts had been studied, and were used to record discoveries made along the way. These markings were interesting both as an indication of how the subject attacked the task, and in their own right as a set of features which might be worth providing in the interface of an on-line tool. As a result, the subjects were encouraged to write on their listing; the listings were collected at the end of each trial and considered part of the collected data.

Based on an evaluation of the data generated by the pilot subject, and a review of the PRL project's needs, a list of questions was constructed for the subjects to focus on while performing the debugging task. A questionnaire was also designed for them to fill out when they finished the task. This questionnaire also became the form around which

PROGRAM DESCRIPTION:

The test program is a morphogenesis simulation, called PROSPER, which loosely models the growth of a colony of bacteria. In PROSPER, the user provides an initial pattern of cells and a collection of production rules which govern their division. Cells are created with division times, and cancer (C) cells are expected to divide more frequently than normal (A) cells. The simulation outputs a trace of the bacteria colony through time.

The default initial pattern of cells looks like this:

```
  A
A C A
  A
```

Sample productions might look like these:

```
        Grow      A
A A A   ----->  A A A
  A               A

  A   Carcinoma   A
A A A   ----->  A C A
  A               A

  A   Metastisis    A
A C A   ----->  A C C A
  A                 A
```

THE BUG REPORT:

The program was started by calling "OUTER-PROSPER" without any arguments. This has the effect of starting the program from the default initial configuration pictured above. The sequence of output frames generated is reproduced below. The problem is that the user expected the productions to cause an explosive growth of cancer cells (cells of type "C"), and instead the A cells grew abundantly.

SAMPLE OUTPUT FROM PROSPER:

```
  A                A                  A
A C A    --->    A C C A    --->    A A C C A    --->
  A                A                  A

Frame 1          Frame 2            Frame 3


  A A              A A                A A A
A A C C A  --->  A A C C A  --->    A A C C A
  A                A A                A A

Frame 4          Frame 5            Frame 6
```

Figure 3-1: Documentation Packet for Program Comprehension Study

all the results were analyzed.

4. THE STANDARD FORM OF THE STUDY

Based on the pilot trial, a basic format was chosen. However, this format was varied in some details over the course of the several trials, in response to comments by the subjects as well as our own observations. This study was not in any sense designed to be a controlled experiment; a controlled situation did not seem as important as one that reflected a realistic scenario yet still allowed the observation and recording of subjects' behavior. For example, when programmers complained that the trials were too long, the allowed time was reduced from an original limit of four hours to two hours.

A small documentation packet was given to subjects at the beginning of the trial. This was the only information the subjects were given about the program and the bug. This packet is shown in Figure 3-1. It describes the purpose of the program and a little about its organization: namely, the program is like a mathematical game called "Life" (with which many programmers are familiar), and it is based on production rules, which specify how the cells reproduce based on their pattern of arrangement. The packet describes the bug and gives an example of the erroneous I/O behavior. The expected behavior of the program was that the cancer cells would reproduce much more quickly than the normal cells. Instead, the normal cells grew, and the cancer cells did not. The error was caused by an improper use of a subroutine that inserted

cells into a priority queue ordered by cell division time.

The instructions for the study are shown in Figure 4-1. The basic task required the subjects to concentrate on finding and fixing the bug in the program. While they worked at this task, they were requested to speak out loud to reveal what they were thinking. It was explained to the subjects what sort of information was desired, and a set of four questions was provided to focus their introspective reports. Those four questions are the central issues around which this study is organized.

As the instructions indicate, data was collected by several means. The primary information came from the subjects themselves as they talked about what they were doing. Though there was a tape recorder running, we primarily relied on the experimenter's notes for data. Occasionally the experimenter interrupted the subjects to remind them of one of the central focus questions, or to query them specifically about their current line of attack.

Following completion of the task, or, more frequently, when time ran out, the subject was given a questionnaire to complete. This questionnaire is shown in Appendix B. It presents the four basic questions for the study, and adds a couple of new questions. The subjects were asked to comment on the experiment itself, and in particular on the issue of supporting documentation. Finally, an annotated list of sample vocabulary for referring to programs was provided, in order to stimulate

INSTRUCTIONS FOR THE EXPERIMENT:

Your task is to find and correct the bug in the program PROSPER. Our interest is in the way in which you go about that task. In order to record your process of exploration and understanding, we want you to feel free to mark up the listing in any way you want. We will also have a tape recorder running, and encourage you to produce a running monologue of your thoughts. From time to time, the experimenter may ask you a question to prod you into revealing what you are thinking about. The experimenter will also be taking notes on what he thinks you are doing.

In particular, we would like you to pay attention to the following sorts of issues and to record comments on them when appropriate:

1) What questions do you ask about the program's structure and design?

2) What sort of vocabulary do you use to refer to objects in the program, and the relations between them?

3) What sort of hypotheses do you construct, and how do you evaluate them?

4) What aids for searching through the program would you like to have?

Figure 4-1: Instructions for Program Comprehension Study

the subjects to generate more new vocabulary.

5. DATA FROM THE STUDY

The study was run with five subjects. They were all experienced programmers, fellow employees at AI&DS. They had varying degrees of familiarity with the particular dialect of Lisp used in the program. None of them had any previous knowledge of the program or of the type of bug it contained.

Of the five subjects, only one actually found the bug in the allotted time. This does not in any way denote failure, since actual debugging was not the focus of our study; rather the emphasis was on observing how the subjects went about studying the program. However, it was apparent that the single subject who found the bug exhibited behavior which differed markedly from the others. All of the subjects spent a lot of time hand simulating the execution of the code; this simulation required extra concentration, and errors that resulted often hindered their efforts. There was substantial difference in the depth to which subjects followed trains of subroutine calls on early passes through the code. The successful subject was the one who was the best at staying at a high level.

Vocabulary was mainly collected from the subjects' verbalizations, and was largely uniform across subjects. Similarly, annotations to the listings, when collected and analyzed, showed relatively consistent patterns across subjects. Additional data was provided by the questionnaires completed by the subjects at the end of each trial.

6. ANALYSIS OF RESULTS

This section presents an analysis of the data in the previous section. In compiling the results of each trial we largely followed the format of the questionnaire, however the analysis of the subject's markings of the program listing was added.

6.1 VARIATIONS IN PROGRAMMERS' STYLES OF PROGRAM UNDERSTANDING

The task we set for the subjects was to find and fix a bug in a program. It was normally assumed that there was only one bug, and that it was fairly well localized. This task should have elicited goal directed understanding; there was no need for the subjects to understand the entire program. It turned out that these programmers varied considerably in their ability to focus in on the problem.

All the subjects adopted a strategy of making a first pass through the program at a high level to get an overview of the program's structure. The plan was to gain a general understanding which would allow the construction of useful hypotheses. Generally they began at the main routine and started tracing through the program's execution to some limited depth. Early on, the focus was on the data structures, and later on, the routines that manipulate them. In the absence of more complete documentation this is a necessary information gathering step.

Familiarity with the program's basic structure is intended to aid generation of hypotheses about the bug. The more specific the hypothesis, the more specific the knowledge needed to form and then test it. In fact there are certain very general classes of bug hypotheses that most programmers will make based on little or no information; experience indicates that these types of mistakes are nearly universal. Examples include passing parameters to a subroutine in the wrong order, and (in Lisp) incorrectly grouping items in parentheses. Checking for these simple but common mistakes can be time consuming, and if done on the first pass, may defeat the plan of performing a quick overview. This approach also allows major features of the program's organization to escape notice for long periods of time.

This is in fact what typically happened to the subjects in this study. Drawn on by the possibility of finding some simple error, most of the subjects tended to push to deeper levels more quickly than they had intended. They often felt they might as well rule out such problems in a section of the code during their first reading of it; by staying in order, they were sure they wouldn't miss anything. Also as one subject noted, the extra study at these lower levels could potentially prove useful later on.

Only the single successful subject really held to his initial plan of performing a high level overview; for the type of bug in this study, this seemed to be the right strategy. The signs of where the problem

might lie were most apparent when taking a global view of the program, because one manifestation of the error was an inconsistent usage of a subroutine call. The successful subject was comparing all the places where a priority queue insertion operation was performed when he noticed this inconsistency.

Much of a well written program is built out of common structures known to all programmers. Following the terminology developed by the Programmer's Apprentice Project at MIT (see [Rich-81] and [Waters-78]), we call such commonly used components "cliches". The bug was caused by the incorrect implementation of a list insertion cliche. Thus, in this case, the understanding of such a cliche was important. The successful subject was in fact explicitly aware of this cliche and of its limitations. In general, cliches are important because they speed understanding by chunking the program into well understood higher level units. An understanding of the specific limitations and likely failure modes of cliches is also a powerful asset in debugging programs.

6.2 VOCABULARY FOR REFERRING TO PROGRAMS

The last question on the questionnaire, which dealt with vocabulary, was often partially ignored; this seems due to the unstructured nature of the question. While most subjects commented on the vocabulary examples given, they rarely added new examples of their own. New voca-

bulary then was most often culled from the experimenter's notes of what the subject said. Analysis of the language used by the subjects in their introspective reports yields five distinct categories of vocabulary. An explanation of each of these classes, with examples, follows.

1. Computer language specific terminology

   For this study the program was written in Lisp and a large number of Lisp specific terms were common in the subjects' speech. Names of particular functions and language keywords are bound to show up, if only to designate locations in the listing, as in the phrase "The second argument of cons". Function names can also be used to designate the concepts they represent in the language. Again, the Lisp function cons creates a new data structure in memory, so a phrase such as "the cons of a and b" refers to a language specific entity. Other language specific concepts or terminology for more general concepts were also evident. Subjects frequently spoke of the binding of variables, a Lisp term for the value of a variable in a certain context.

2. General programming cliches

   The program, despite its disorganization, lapses of style, and lack of documentation, was largely constructed out of commonly used programming abstractions (cliches). It made heavy use of a hash table abstraction and of a priority queue. These were fairly obvious in the code and were noticed eventually by all the subjects. Each such cliche comes with some vocabulary commonly used to refer to its parts and the operations defined for it. Simple examples include insertion and deletion. The subjects did in fact use this terminology when talking about parts of these cliches.

3. Domain specific terminology

   The domain of the study program was a colony of cells, some of them cancerous, growing in some environment. While not experts in cell biology, all the subjects developed some reasonable expectation of the program's behavior based on their understanding of the domain. As with the language specific vocabulary, many of these terms appear as function names, in this case defined and later used in the program, and similarly as variable names. Subjects frequently found occasion to talk about cells and metastasis. Potentially an even richer source of such terminology is the documenta-

tion that should accompany a program.

4. Natural Language Constructs

In natural language, objects may be referenced in a number of indirect ways. Anaphora designates references to objects previously mentioned in the discourse. Definite noun phrases or pronouns may serve this purpose. For example, subjects frequently designated an argument to a function as "...its first argument." The "it" refers to the function.

Deixis designates references to objects present in the environment, either by pointing or description. Such references are more common when trying to make clear to someone else which object you are referring to. Deixis was accordingly less common in this study, as the subjects felt they were primarily talking to themselves.

5. Idiosyncratic, user-specific views of the world

It was apparent that the subjects developed different models of the program varying along idiosyncratic lines. For example, one subject viewed the several different internal representations of the cell colony as successive projections (in a mathematical sense) of the basic representation, which he took to be the events queue. He used this terminology to talk about the data structures and the algorithms that mapped between them.

6.3 ANALYSIS OF PROGRAM LISTING ANNOTATIONS

The listings given to the subjects for study were collected at the end of each trial and analyzed in order to determine what sorts of interface facilities should be made available to a user in an on-line tool. Both graphic and textual annotations were common.

The major classes of graphic annotations were highlighting, grouping, and connecting. Highlighting was used to focus attention on a part of the listing, or to make it easier to find again in the future.

Subjects made use of multiple colors, and underlined, boxed or shaded the desired object. Grouping generally consisted of drawing a box or brackets around some items in the listing to identify them as a cohesive unit. Frequently, text was attached to explain the significance of the grouping. Connecting was usually done with an arrow, either between objects in the listing, or between an object and a textual comment added by the subject to describe the object.

Text was used to record any discoveries the subject deemed worth remembering. This included both labels and longer comments or explanations. It was scribbled wherever there was space, and connected to some designated object in the listing.

6.4 DESIRED DOCUMENTATION

The general attitude of the subjects towards documentation can best be summed up by a comment one of them made on the questionnaire: "I know of only one type of documentation that is not especially helpful: wrong documentation." The only real concern any subject expressed was that the programmer might be overloaded with irrelevant information. Many types of documentation were suggested by the subjects, the most novel perhaps being detailed history information, including answers to such questions as: who wrote it, when was it written, how was it tested, did it ever work before, were existing subroutine libraries used, etc. A good knowledge of this type of history can strongly influence what type

of bugs are suspected.

6.5 DESIGN OF THE STUDY

In order to allow fine tuning of the study, a question about the format of the study itself was included on the questionnaire. After each trial, the recommendations were considered, and slight alterations were sometimes made to the study. Our major concern was that the subjects find the experience as "natural" as possible.

The principle findings here were that the subjects had no problem understanding the task or performing it with an observer present. The need to talk out loud while studying the program was not viewed as a significant inconvenience, and in general, subjects felt they performed as they would have given a comparable real world task.

The major caveat to this appraisal was that normally the subjects would expect to have better tools. In particular, real debugging would not get very far without a run time environment. A large effort was required by the subjects to perform hand simulations of the code, and the errors they made in the process complicated the effort of finding the mistake in the program (as well as straining their ability to concentrate on the task at hand). Even without the facilities to run programs, subjects would have greatly appreciated a standard text editor with its basic string search capabilities.

Subjects also felt that the questionnaire was not sharply enough focused, a problem which we felt derived from the exploratory nature of the study. The length of the questionnaire had its repercussions, for instance, few of the subjects gave interesting responses to the final question on vocabulary.

7. IMPLICATIONS FOR THE PRL/EPM

This study has several implications for the design of both the PRL and the EPM. Some of these are confirmations of assumptions and biases we have been working with since the start of the project; others are genuinely new issues raised by the performance of the subjects in this test.

In the confirmation category there were two major observations:

1. The multiple views of the EPM are useful.

   The subjects really did look for information at all the levels from simple text string searches up to searches through all instances of some cliche action. Examples of PRL operations they performed by hand include: "Visit in sequence all the functions called from this function," "Highlight all the exits from this loop," and "Visit all the places Event-Queue has its value changed."

2. Documentation is critically important.

   The lack of documentation in this study highlighted the importance of this information source. Even the small packet provided was a major improvement over the pilot trial where there was no documentation at all. Some of the subjects' specific requests for information, such as about the history of the program, could reasonably be kept available as documentation, easing the task of debugging considerably. Making all the information available online would clearly be a major advance. These are issues we are considering both in the IPE project and in a separate project called the "Documentation Assistant".

Though not particular surprising, these results are relevant to our efforts, and tend to support the assumptions on which we have based much of work. There were four areas where this study yielded new results:

1. A sampling of vocabulary for the PRL was gathered.

   Of the new results from this study, the most important, and the one that most directly motivated the study in the first place, was the sampling of programmer vocabulary. The natural vocabulary we gathered turned out to be drawn from the five distinct categories presented earlier: computer language specific terminology, computer programming cliches, domain specific terminology, natural forms of reference, and user specific views of the world.

   Of these five categories of vocabulary, the EPM directly provides representations for the first three: the syntax representation of the EPM provides computer language specific terminology; the typical programming pattern representation provides terminology for programming cliches; and intentional aggregates provide domain specific terminology. The remaining two categories are not directly addressed in the current EPM design. To allow the user full freedom of expression would require the PRL to deal with all the intricacies of natural language processing (a currently unsolved problem); moreover, when using a keyboard to enter queries and commands, it is not clear that a user wants to type out full sentences (or even sentence fragments). The PRL would require a very specialized user model to allow users to talk about the program in their own highly stylized way.

2. Individual differences imply need for customization.

   The study showed some of the ways that individual programmers vary in work style. To support programmers effectively, it appears necessary to provide for customization of the work environment. An intelligent programming environment might maintain a user profile either based on explicit user requests, or in an advanced system, based on autonomous observations. While some existing editors allow a small amount of individual user control over the behavior of some features, none have extensive user models.

3. Context-sensitive information management is important.

   There are three major open questions on this issue. When should information be available but hidden? Subjects indicated a desire for many types of documentation, but they did not want to see all of them all of the time. When should information be ellided? In order to fit on the screen, code and documentation may

have to be condensed. When should information be forgotten? While much of what subjects wrote on their listings was intended to be permanent, sometimes they made assumptions or drew conclusions which they later wanted to change. They also frequently made notations that were only intended to be temporary reminders of some postponed task.

4. Useful user interface features were identified.

The study pointed out the need for sophistication in the user interface. Information must be managed not only internally, but also in its presentation on the screen. The user should be allowed to work in the familiar paper and pencil mode if and when appropriate, and should be able to call up all relevant documentation online, but cannot afford to be overwhelmed by cluttering the screen with everything the system has stored about some piece of a program.

8. SUMMARY

The key results of this study were (1) a confirmation of the usefulness of the conceptual mechanisms provided by the Extended Program Model, and (2) identification of new areas and issues important for the development of the PRL and EPM.

The new issues outline a program of further work to pursue in the continuation of the PRL project. The vocabulary lists have already spurred the development of a tentative formal syntax for queries in PRL. Further analysis will lead to refinement of this specification, and eventually to an extended and modified version which will define the user language. Observations about how practicing programmers go about understanding programs will provide significant guidance on future PRL work. Other new issues will influence design work on the IPE, which continues under a separate contract.

9. PLANS FOR FURTHER DEVELOPMENT

Further work on the PRL will concentrate on specifying the formal syntax, complete basic vocabulary, and external user syntax for the language. The vocabulary data from this study provide a good starting point for such an effort. We plan to define a relatively simple, strongly constrained syntax for the system's internal use, while providing a looser, more forgiving syntax for the user. Given these two levels of the language, we must design a method for mapping between them.

There are several issues highlighted by this study that we will not pursue. The first of these is the need for strong support in the debugging task, ideally in the form of a dynamic debugging environment. We also do not plan to tackle head-on the problem of processing unconstrained natural language which is a large area of research that is not directly related to the PRL. Finally, we do not intend to model each individual user so as to understand their personal idiosyncratic vocabularies.

We remain uncommitted as to how much of the high level modeling of users and domains we will be able to handle. These have the potential for significant payoff, and are the most likely areas for introducing additional intelligence into the system, beyond its basic understanding of the programming domain itself. Such modeling is, like natural

language, a major research effort in its own right, but one which is more directly germane to the goal of developing intelligent aids to software comprehension.

10. PERSONNEL

10.1 PERSONNEL

The Program Reference Language (PRL) research project is being performed within the User Aids Program of AI&DS, with Dr. Brian P. McCune, Program Manager, as Principal Investigator. Other members of the AI&DS technical staff who are contributors to the project include Jeffrey S. Dean, Eric A. Domeshek, Michael A. Brzustowicz, and Daniel G. Shapiro.

Dr. Brian P. McCune is the Principal Investigator of the PRL project. He received his Ph.D. in Computer Science from Stanford University in 1979; the title of his thesis was "Building Program Models Incrementally from Informal Descriptions." During the past decade, Dr. McCune has done research in the areas of artificial intelligence, software systems, and computer architecture, with emphasis on artificial intelligence approaches to software development and maintenance, information retrieval, database management, hypothesis formation, planning, and distributed processing. He has been the principal investigator of research projects to select and design candidate AI tools for assisting in the maintenance of ADA programs (sponsored by Rome Air Development Center), to design an intelligent program editor for ADA, to determine

the feasibility of automatically generating operating systems, and to design and implement a knowledge-based system for textual information retrieval. Dr. McCune is an Associate Editor of The AI Magazine. He has been invited to discuss the application of artificial intelligence to defense problems numerous times, both at workshops and in published papers.

Jeffrey S. Dean has recently begun to play a key role in the PRL project; he is currently leading the related Intelligent Program Editor project, and was previously the leader of the AI&DS Software Maintenance Project, which defined advanced Ada tools for software maintenance. He received his Masters degree in Computer Science/Computer Engineering from Stanford University, where he worked on the automatic derivation of operating systems. His main research interest is the application of AI to software tools. He came to AI&DS in January 1981 from Bell Telephone Laboratories, where he was involved in the development and maintenance of the UNIX operating system and its utilities.

Daniel G. Shapiro has been contributing to the PRL project since joining AI&DS in October 1981, after receiving a Masters degree in electrical engineering and computer science from the Massachusetts Institute of Technology. His research interests include artificial intelligence, expert systems, and software engineering. At AI&DS he has done work on expert systems for program and documentation editing, information retrieval, and mission planning. His masters thesis, enti-

tled "Sniffer: A system that Understands Bugs," involved the design and implementation of a semantics-based debugger for the Programmer's Apprentice project at the MIT Artificial Intelligence Laboratory. He also taught software engineering courses at MIT.

Eric A. Domeshek was responsible for much of the PRL experiment which studied how people think about programs. Mr. Domeshek received an A.B. in Physics from Harvard College. His course work also emphasised computer science and cognitive science. His technical interests are in Artificial Intelligence, particularly knowledge representation, and in computer graphics.

Michael A. Brzustowicz has been involved with the PRL project since joining AI&DS in November 1983. He received an S.B. degree in Physics from the Massachusetts Institute of Technology in 1979 and received his M.S.E.E. in Computer Engineering from Carnegie-Mellon University in 1980; his thesis work was entitled "A System for the Implementaiton of Models of Reasoning with Uncertain Data." Mr. Brzustowicz's current areas of interest include Artificial Intelligence, Software Engineering, Ergonomic User Interfaces, and Computer Aided Processes. Prior to joining AI&DS, Mr. Brzustowicz worked for the Development Systems Software Group of the Semiconductor Division of Texas Instruments, and for the Unix Development Group at Bell Laboratories.

10.2 INTERACTIONS

Dr. Brian P. McCune is an Associate Editor of The AI Magazine, the publication of the American Association for Artificial Intelligence. He is on the Editorial Advisory Board of Defense Electronics and also The Artificial Intelligence Report.

Dr. McCune was an invited speaker to COMPSAC '83 (November 1983) and EASCON '83 (September 1983), and was an invited participant to Knowledge Based Software Assistant Workshop at AAAI-83 (August 1983). He attended the NAVAIR/ONR Aviation Software Workshop (October 1983), the DARPA Formalized Software Development Workshop (November 1983), the Conference on Inference Theory and AI (November 1982), and the Software Maintenance Workshop (December 1983).

In addition to lectures associated with papers that appeared in published conference proceedings, project staff members have given numerous lectures around the country. Dr. McCune has been lecturing throughout the federal government on software maintenance and intelligence problems and the potential of artificial intelligence to help solve them. Along with Daniel G. Shapiro, he presented results from the PRL project to Dr. Northrup Fowler III and Douglas White of RADC at AI&DS in December 1982.

Dr. McCune was one of twelve technologists selected to participated in the Government-sponsored Conference on Inference Theory and Artificial Intelligence, held in Leesburg, Virginia, in November 1982 to discuss how artificial intelligence, decision analysis, and inference theory might be combined to enhance the production of intelligence. Dr. McCune attended the DoD Software Initiative Workshop in Raleigh, North Carolina, in February 1983.

Dr. McCune attended the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83), held in Karlsruhe, Germany, in August 1983; the National Conference on Artificial Intelligence (AAAI-83), Washington D.C., August 1983; and the Symposium on Intelligence Applications of Advanced Computer and Information Technology: Focus on Artificial Intelligence, sponsored by the Offices of Research and Development and Scientific and Weapons Research, Central Intelligence Agency, and held in Washington, D.C., November 1982.

Dr. McCune has been interfacing heavily with both operational and developmental commands in the Air Force and elsewhere in DoD and industry in order to understand current and future problems of software development and maintenance. Within the Air Force, Dr. McCune has met with personnel at the Air Force Office of Scientific Research, Rome Air Development Center, Wright Aeronautical Laboratories, Foreign Technology Division, Strategic Air Command headquarters, Air Force Communications Computer Programming Center, and Air Force Satellite Control Facility.

Elsewhere in DoD he has talked with the Defense Intelligence Agency, Office of the Undersecretary of Defense for Research and Engineering, Defense Advanced Research Projects Agency, DoD STARS Program, ADA Joint Program Office, Office of Naval Research, Naval Electronics Systems Command, Naval Sea Systems Command, Naval Intelligence Command, Naval Research Laboratory, Naval Ocean Systems Center, Naval Intelligence Center, Naval Weapons Center, Army Research Office, Army Center for Tactical Computer Systems, and Army Ballistic Missile Defense Advanced Technology Center.

Dr. McCune has also visited numerous universities and research centers to assess the state of the art in automatic programming at first hand. Places visited include Harvard University, Massachusetts Institute of Technology, Carnegie-Mellon University, Duke University, University of California at Irvine, and Stanford University.

Jeffrey S. Dean presented a paper on a study of software maintenance at the Software Maintenance Workshop (December 1983). He attended the Symposium for Application and Assessment of Automated Tools for Software Development (November 1983); AAAI-83; IJCAI-81; the Working Conference on Automated Tools for Information Systems Design and Development, held in New Orleans in January 1982 and sponsored by IFIP Working Group 8.1 on Desigr and Evaluation of Information Systems; and UNICOM, the semiannual UNIX users' conference (January 1983).

Daniel G. Shapiro was a panelist at the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, held in Pacific Grove, California, in March 1983. He presented papers on the PRL at the IEEE Trends and Applications Conference (May 1983) and the Seventh International Conference on Software Engineering (March 1984). He presented papers on information retrieval at AAAI-83 and IJCAI-83.

Eric A. Domeshek attended the Symposium for Application and Assessment of Automated Tools for Software Development (November 1983) and AAAI-83.

Michael A. Brzustowicz attended the Symposium for Application and Assessment of Automated Tools for Software Development (November 1983).

10.3 PUBLICATIONS

Members of PRL project staff have published a number of papers; reprints of those papers most relevant to the PRL project have been included as Appendices to this proposal. A cumulative chronological list of publications appearing in technical journals and conference proceedings is listed below:

Daniel G. Shapiro, Jeffrey S. Dean, and Brian P. McCune, "A Knowledge Base for Supporting an Intelligent Program Editor," 7th International Conference on Software Engineering, March 1984.

Andrew S. Cromarty, Daniel G. Shapiro and Michael R. Fehling, "Still Planners Run Deep: Shallow Reasoning for Fast Replanning," Proceedings, Society of Photo-Optical Instrumentation Engineers, Technical Symposium East, 1984, to appear.

Jeffrey S. Dean and Brian P. McCune, "An Informal Study of Software Maintenance Problems," Proceedings, Software Maintenance Workshop, December 1983.

Brian P. McCune and Jeffrey S. Dean, "Trends for Advanced Software Tools," Defense Science 2001+ (reprint of EASCON '83 paper), December 1983.

Brian P. McCune, Richard M. Tong, Jeffrey S. Dean, and Daniel G. Shapiro, "RUBRIC: A System for Rule-Based Information Retrieval," Proceedings, COMPSAC 1983, November 1983.

Brian P. McCune and Jeffrey S. Dean, "Trends for Advanced Software Tools," invited paper, Proceedings, EASCON '83, September 1983.

Richard M. Tong, Daniel G. Shapiro, Brian P. McCune, and Jeffrey S. Dean, "A Rule-Based Approach to Information Retrieval: Some Results and Comments," Proceedings, National Conference on Artificial Intelligence, Washington, D.C., August 1983.

Richard M. Tong, Daniel G. Shapiro, Jeffrey S. Dean, and Brian P. McCune, "Performance Analysis of a Rule-Based Information Retrieval System," 1983 National Conference on Artificial Intelligence, Washington, D.C., August 1983.

Richard M. Tong, Daniel G. Shapiro, Jeffrey S. Dean, and Brian P. McCune, "A Comparison of Uncertainty Calculi in an Expert System for Information Retrieval," Eighth International Joint Conference on Artificial Intelligence, Karlsruhe, West Germany, August 1983.

Brian P. McCune and Robert J. Drazovich, "Radar with Sight and Knowledge," invited paper, Defense Electronics, August 1983.

Richard M. Tong and Daniel G. Shapiro, "An Experiment with Multiple Valued Logics in an Expert System," Proceedings of the IFAC Symposium on Fuzzy Information, Knowledge Representation and Decision Analysis, Marseille, France, July 1983.

Daniel G. Shapiro and Brian P. McCune, "The Intelligent Program Editor: A Knowledge-Based System for Supporting Program and Documentation Maintenance," Proceedings of the Trends and Applications Conference of the IEEE, May 1983.

Gerald Wilson, Eric A. Domeshek, Ellen L. Drascher, and Jeffrey S. Dean, "The Multipurpose Presentation System," Proceedings, Very Large Data Base Conference, 1983.

Jeffrey S. Dean and Brian P. McCune, Advanced Tools for Software Maintenance, Rome Air Development Center, RADC-TR-82-313, December

1982.

Brian P. McCune, Jeffrey S. Dean, Daniel G. Shapiro, and Richard M. Tong, "Rule-Based Information Retrieval," Workshop on Intelligence Applications of Advanced Computer and Information Technology: Focus on Artificial Intelligence, Office of Research and Development, Office of Scientific and Weapons Research, Central Intelligence Agency, Washinton, D.C., November 1982.

Robert J. Drazovich, Brian P. McCune, and J. Roland Payne, "Artificial Intelligence: An Emerging Military Technology," invited paper, Conference Record, EASCON '82: Fifteenth Annual Electronics and Aerospace Systems Conference, Institute of Electrical and Electronics Engineers, Inc., Washington, D.C., September 1982, Pages 341-348.

Brian P. McCune, editor, "AI at AI&DS," The AI Magazine, Volume 2, Number 2, Summer 1981, pages 44-47.

Daniel G. Shapiro, "Sniffer: A System that Understands Bugs," MIT/AIM/638, June 1981.

Brian P. McCune, "Incremental, Informal Program Acquisition," Proceedings of the First Annual National Conference on Artificial Intelligence, Stanford University, Stanford, California, August 1980, pages 71-73.

Daniel G. Shapiro, "A Proposal for Sniffer, A System that Understands Bugs," MIT/AI Working Paper 202, July 1980.

Cordell Green, Richard P. Gabriel, Elaine Kant, Beverly I. Kedzierski, Brian P. McCune, Jorge V. Phillips, Steve T. Tappel, and Stephen J. Westfold, "Results in Knowledge-Based Program Synthesis," IJCAI-79: Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Volume 1, Computer Science Department, Stanford University, Stanford, California, August 1979, pages 342-344.

George R. Lewis, J. Shirley Henry, and Brian P. McCune, "The BTI 8000: Homogeneous, General-Purpose Multiprocessing," in Richard E. Merwin, editor, 1979 National Computer Conference, AFIPS Conference Proceedings, Volume 48, AFIPS Press, Montvale, New Jersey, June 1979, pages 513-528.

Cordell Green and Brian P. McCune, "Knowledge-Based Programming Applications," Applications of Image Understanding and Spatial Processing to Radar Signals for Automatic Ship Classification: Proceedings of a Workshop, Naval Electronic Systems Command, Washington, D.C., February 1979, pages 94-99.

Cordell Green and Brian P. McCune, "Application of Knowledge-Based Programming to Signal Understanding Systems," Distributed Sensor

Nets: Proceedings of a Workshop, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, December 1978, pages 115-118.

Brian P. McCune, "The PSI Program Model Builder: Synthesis of Very High-Level Programs," Proceedings of the Symposium on Artificial Intelligence and Programmin Languages, SIGPLAN Notices, Volume 12, Number 8, SIGART Newsletter, Number 64, August 1977, pages 130-139.

APPENDIX A

Lisp Program Used in PRL Study

This appendix contains a listing of the Lisp program used in the PRL study.

11. REFERENCES

1. Rich, Charles, "Inspection Methods in Programming", AI-TR-604, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., 1981.

2. Shapiro, Daniel G., "Sniffer: A System that Understands Bugs", AIM-638, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., 1981.

3. Shapiro, Daniel G., McCune, Brian P., "Searching a Knowledge Base of Programs and Documentation", AI&DS TM-1014-2, January 1983.

4. Shapiro, Daniel G., McCune, Brian P., "A Knowledge Based System for Supporting Program and Documentation Maintenance", Proceedings, Trends and Applications, 1983, pp. 226-232.

5. Shapiro, Daniel G., Dean, Jeffrey S., and McCune, Brian P., "A Knowledge Base for Supporting an Intelligent Program Editor", Proceedings, 7th International Conference on Software Engineering, 1984, to appear.

6. Waters, Richard C., "Automatic Analysis of the Logical Structure of Programs", AI-TR-492, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., 1978.

```
(declare
 (special events-queue div-time grid transform-lib directions
north south east west prosper-display-buffer))

(defun outer-prosper (&optional (evq (make-an-evq)))
  (create-prosper-display-buffer)
  (prosper evq))

(defun prosper (events-queue)
  ((lambda (transform-lib grid)
    (prog (matches cell div-time)
          (grid-init events-queue grid)

      foo
          (cond ((null events-queue) (return nil)))
          (display-grid grid)
          (setq cell (top-cell events-queue))
          (setq div-time (top-time events-queue))
          (setq events-queue (rest events-queue))
          (setq matches (find-transforms cell transform-lib))
          (apply-transforms matches cell grid)
          (go foo)))

   (create-transform-lib)
   (create-grid)))

;the def for metastasize can be snarfed off of the plans ;as well as the
def for events-queue-insert

(defun grid-init (evq grid)
  (do ((q evq (rest q))
       (topc nil))
      ((null q) nil)
    (setq topc (top-cell q))
    (ht-insert (cell-location topc) topc grid)))

(defun create-grid ()
  (let ((g (array g t 21)))
    g))

(defun top-time (evq) (cond ((null evq) nil) ((car (car evq))))) (defun
top-cell (evq) (cond ((null evq) nil) ((cdr (car evq))))) (defun rest
(evq) (cond ((null evq) nil) ((cdr evq))))

(defun find-transforms (key-cell tlib)
   (ht-lookup-all (cell-type key-cell) tlib))

(defun apply-transforms (candidates key-cell grid)
  ;run the filter function associated with each candidate.
  ;if it succeeds, apply the transform to the bindings returned by the
```

```
  ;filter function only run the first matching transform
  (do ((trs candidates (cdr trs))
       (transform nil)
       (bindings nil))
      ((null trs) nil)

    (setq transform (car trs))
    (setq bindings (apply-selector (car transform) key-cell grid))
      ;bindings is a list of cells which form the context for the key
cell
      ;ALL selector functions have the key-cell as the last element of
the
      ;bindings returned
    (cond (bindings (return (apply (car (cdr transform)) bindings))))))

(defun apply-selector (selector-fun key-cell grid)
  ;takes care of mapping the selector function  all possible ways
  ;onto the grid
  (do ((nominal-north (+ 1 (random 4)) (+ 1 nominal-north))
       (tries 1 (+ 1 tries))
       (win nil))
      ((> tries 4) nil)
    (let ((north (nth nominal-north directions))
          (west (nth (+ nominal-north 1) directions))
          (south (nth (+ nominal-north 2) directions))
          (east (nth (+ nominal-north 3) directions)))
      (and (setq win (funcall selector-fun key-cell grid))
           (return win)))))

(defun north (loc) (loc+ north loc)) (defun south (loc) (loc+ south
loc)) (defun east (loc) (loc+ east loc)) (defun west (loc) (loc+ west
loc))

(defun north-cell (c g) (ht-lookup (north (cell-location c)) g)) (defun
south-cell (c g) (ht-lookup (south (cell-location c)) g)) (defun east-
cell (c g) (ht-lookup (east (cell-location c)) g)) (defun west-cell (c
g) (ht-lookup (west (cell-location c)) g))

(defun display-grid (grid)
  (clear-prosper-display-buffer)
  (map-over-all-ht-datums grid 'place-cell-in-buffer)
  (print-prosper-display-buffer))

(defun events-queue-insert (item time evq)
   (prog (nq oq entry)
         (setq entry (cons time item))
         (cond ((or (null evq) (before? entry (car evq)))
                (return (cons entry evq))))
         (setq nq (cdr evq))
         (setq oq evq)
```

```
    lp   (cond ((or (null nq) (before? entry (car nq)))
               (rplacd oq (cons entry nq))
               (return evq)))
         (setq oq nq)
         (setq nq (cdr nq))
         (go lp)))

(defun before? (item1 item2)
  (< (car item1) (car item2)))

(defun make-cell (type loc divc) (copy-the-damn-thing (list type loc
divc))) (defun cell-type (cell) (car cell)) (defun division-count (cell)
(car (cdr (cdr cell)))) (defun cell-location (cell) (car (cdr cell)))
(defun change-cell-type (cell new-type) (rplaca cell new-type)) (defun
increment-division-count (cell)
  (rplaca (cdr (cdr cell)) (+ 1 (division-count cell)))) (defun create-
cancer-cell () (copy-the-damn-thing '(c () 1)))

(defun make-location (x y) (list x y)) (defun loc-x (loc) (car loc))
(defun loc-y (loc) (car (cdr loc))) (defun loc+ (l1 l2) (make-location
(+ (loc-x l1) (loc-x l2))
                                   (+ (loc-y l1) (loc-y l2)))) (defun
loc- (l1 l2) (make-location (- (loc-x l1) (loc-x l2))
                                   (- (loc-y l1) (loc-y l2))))

;;;;;;;;;;;;;;; HASH TABLE ABSTRACTION ;;;;;;;;;;;;;;; ; this hash table
indexes K-enitities (representing monocells) by K-datums ; which
represent cell locations

(defun ht-lookup (key ht)
  ;find the appropriate bucket, then search it for a tag which
  ;is K-equal key.  Return the cdr of the alist element if one is found.

  (let ((bucket (ht (bucket-select key ht)))
        (item nil))
    (setq item (assoc key bucket))
    (and item (cdr item))))

(defun ht-lookup-all (key ht)
 (let ((bucket (ht (bucket-select key ht))))
   (cond ((null bucket) nil)
         ((mapcar
            '(lambda (x) (cond ((equal (car x) key) (cdr x))))
           bucket)))))

(defun ht-delete (key ht)
  ; Find the item (a monocell) in the bucket indexed by (k-qual key).
  ; Splice it out if it is there.
  (let ((bucket-num (bucket-select key ht))
        (bucket nil))
```

```
    (setq bucket (ht bucket-num))
    (cond ((and bucket (k-equal (caar bucket) key))
           (store (ht bucket-num) (cdr bucket)))
          ((do ((old bucket new)
                (new (cdr bucket) (cdr new)))
               ((null new) nil)
             (and (k-equal (caar new) key)
                  (return (rplacd old (cdr new)))))))
    key))

(defun ht-insert (key item ht)
  (let ((bucket-num (bucket-select key ht))
        (pair (cons key item)))
    (store (ht bucket-num) (cons pair (ht bucket-num)))
    key))

(defun bucket-select (key ht)
  (remainder (sxhash key) 21))

(defun sxhash (key)
  (apply '+ (exploden key)))

(defun ht-dump (ht)
  (do ((i 0 (+ 1 i)))
      ((> i 20))
    (print (ht i))
    (terpri)))

(defun map-over-all-ht-datums (ht fun)
  (do ((i 0 (+ i 1)))
      ((> i 20))
   (mapcar '(lambda (x) (funcall fun (cdr x))) (ht i))))

;(defun ht-create () ;  (let ((g (array nil t 21))) ;    g))

;STUB ALERT (defun k-equal (a1 a2) (equal a1 a2)) ;a stub

;;;;;; the other prosper functions ;;;;;;;

(defun make-room-between (c1 c2 g)
  (let ((addend (loc- (cell-location c2) (cell-location c1))))
    (push-out c2 addend g)))

(defun push-out (cell addend grid)
  (let ((new-loc (loc+ (cell-location cell) addend)))
    (cond ((null (ht-lookup new-loc grid)) (grid-insert cell new-loc
grid))
          (t (push-out (ht-lookup new-loc grid) addend grid)
             (grid-insert cell new-loc grid)))))
```

```
(defun grid-insert (cell location grid)
  ;remove the cell from its old location in the grid,
  (ht-delete (cell-location cell) grid)

  (ht-delete location grid) ;remove whatever cell currently lurks at
location

  ;side effect the cell!
  (move-cell cell location grid))

(defun move-cell (cell loc grid)
  ;just blithely crams the new fella into the grid
  ;side effects the cell
  (rplaca (cdr cell) loc) ;  (k-eval '(rplaca (cdr ,cell) loc))
  (ht-insert loc cell grid))

;(defun make-location (x y) ;  (k-cons x y))

;(defun loc-y (location) (k-car (k-cdr location))) ;(defun loc-x
(location) (k-car location))

(defun make-an-evq ()
  (copy-the-damn-thing
   '((0 c (0 0) 1) (3 a (0 1) 1) (3 a (0 -1) 1) (3 a (1 0) 1) (4 a (-1
0) 1))))

(defun copy-the-damn-thing (thing)
  (cond ((dtpr thing)
         (cons (copy-the-damn-thing (car thing))
               (copy-the-damn-thing (cdr thing))))
        (thing)))

(defun create-prosper-display-buffer ()
   (defprop prosper-display-buffer 15 width)
   (defprop prosper-display-buffer 15 height)
   (setq prosper-display-buffer (array prosper-display-buffer t
                                       (get 'prosper-display-buffer
'width)                                        (get 'prosper-display-
buffer 'height))))

(defun clear-prosper-display-buffer (&aux width height)
   (fillarray prosper-display-buffer (list '| |)))

(defun place-cell-in-buffer (cell &aux (cell-loc (cell-location cell))
                                  width height x-pos y-pos)
  (setq width  (get 'prosper-display-buffer 'width))
  (setq height (get 'prosper-display-buffer 'height))
  (setq x-pos (+ (loc-x cell-loc) (fix (/ width 2))))
  (setq y-pos (+ (loc-y cell-loc) (fix (/ height 2))))
  (store (prosper-display-buffer y-pos x-pos) (cell-type cell)))
```

```
(defun print-prosper-display-buffer (&aux width height)
  (setq width  (subl (get 'prosper-display-buffer 'width)))
  (setq height (subl (get 'prosper-display-buffer 'height)))
  (do i 0 (addl i) (>& i height)
      (terpri)
      (do j 0 (addl j) (>& j width)          (princ (prosper-display-
buffer i j))))
  (terpri))

;;;;;;;;; transforms and selector functions ;;;;;;;; (defun a-cell-
with-room-to-grow (key-cell grid)
  (let ((north-neighbor (north-cell key-cell grid)))
    (and (eq (cell-type key-cell) 'a)
         (null north-neighbor)
         (list (north (cell-location key-cell)) key-cell))))

(defun grow-A-cell (empty-location key-cell)
  (increment-division-count key-cell)
  (prog (new-cell)
     (setq new-cell (make-cell 'a empty-location 1))
     (grid-insert new-cell(cell-location new-cell) grid)
     (setq events-queue            (events-queue-insert new-cell (+
div-time 5) events-queue))
     (setq events-queue            (events-queue-insert key-cell (+
div-time 5) events-queue))))

(defun gotta-b-neighbor (key-cell grid)
  (let ((north-neighbor (north-cell key-cell grid)))
    (and (eq (cell-type north-neighbor) 'b) (list key-cell))))

(defun age-prematurely (key-cell)
  (increment-division-count key-cell)
  (increment-division-count key-cell)
  (increment-division-count key-cell)
  (setq events-queue         (events-queue-insert key-cell (+ div-time
3) events-queue)))

(defun surrounded-by-A-cells (key-cell grid
                              &aux (key-loc (cell-location key-cell)))
  ;a filter function for the carcinoma transform
  ;returns a list of cells which are the context for the metast
transform
  (let ((tc (north-cell key-cell grid))
        (bc (south-cell key-cell grid))
        (rc (east-cell key-cell grid))
        (lc (west-cell key-cell grid)))
    (and
      (not (eq (cell-type key-cell) 'c))
      tc (eq (cell-type tc) 'a)
      bc (eq (cell-type bc) 'a)
```

```
      rc (eq (cell-type rc) 'a)
      lc (eq (cell-type lc) 'a)
      (list key-cell))))

(defun carcinoma (key-cell)
  (increment-division-count key-cell)
  (change-cell-type key-cell 'c)
  (setq events-queue         (events-queue-insert key-cell (+ div-time
1) events-queue)))

(defun enclosed-cancer-cell (key-cell grid)
  (let ((tc (north-cell key-cell grid))
        (bc (south-cell key-cell grid))
        (rc (east-cell key-cell grid))
        (lc (west-cell key-cell grid)))
    (and
      (eq (cell-type key-cell) 'c)
      tc
      bc
      rc
      lc
      (list tc key-cell))))

(defun cancer-cell-with-one-neighbor (key-cell grid)
  (let ((buddy (west-cell key-cell grid)))
    (and buddy (list buddy key-cell))))

(defun metastasize (right-cell key-cell)
  ((lambda (new-cell location)
     (increment-division-count key-cell)
     (make-room-between key-cell right-cell grid)
     (grid-insert new-cell location grid)
     (events-queue-insert new-cell (+ div-time 2) events-queue)
     (events-queue-insert key-cell (+ div-time 2) events-queue))
   (create-cancer-cell) (cell-location right-cell)))

(defun old-aged-cell (key-cell grid)
  (and (> (division-count key-cell) 4) (list key-cell)))

(defun die (key-cell)
  (ht-delete (cell-location key-cell) grid))

(defun cancer-cell-filter (key-cell grid)
  (cond ((eq (cell-type key-cell) 'c) (list key-cell))))

(defun cancer-cells-never-die (key-cell)
  (setq events-queue         (events-queue-insert key-cell (+ div-time
5) events-queue)))

(defun create-transform-lib (&aux tl)
```

```
  (setq tl (array tl t 21)) ;  (ht-insert 'b (list 'surrounded-by-A-
cells 'carcinoma) tl)
  (ht-insert 'a (list 'surrounded-by-A-cells 'carcinoma) tl)
  (ht-insert 'a (list 'a-cell-with-room-to-grow 'grow-A-cell) tl)
 ; (ht-insert 'f (list 'surrounded-by-A-cells 'carcinoma) tl) ;  (ht-
insert 'c (list 'gotta-b-neighbor 'age-prematurely) tl) ;  (ht-insert 'c
(list 'cancer-cell-filter 'cancer-cells-never-die) tl)
  (ht-insert 'c (list 'cancer-cell-with-one-neighbor 'metastasize) tl) ;
(ht-insert 'c (list 'enclosed-cancer-cell 'metastasize) tl)
  tl)

;  (ht-insert 'c (list 'old-aged-cell 'die) tl)


;;unused transforms ;;;

(defun cees-abound (key-cell grid)
  (let ((neighbor (west-cell key-cell grid)))
    (and neighbor (eq (cell-type neighbor) 'c) (eq (cell-type key-call)
'c)
         (list neighbor
               key-cell))))

;;;;; globals ;;;;;;; (setq directions (list
                    (make-location 0 1)
                    (make-location -1 0)
                    (make-location 0 -1)
                    (make-location 1 0))) (rplacd (last directions)
directions) (setq north (nth 1 directions)
      west (nth 2 directions)
      south (nth 3 directions)
      east (nth 4 directions))

(defun block-a-cells ()
  (copy-the-damn-thing
    '((10 c (0 0) 1) (24 a (0 1) 2) (24 a (0 -1) 2) (24 a (-1 0) 2)
      (24 a (1 0) 2) (24 a (1 1) 2) (24 a (1 -1) 2) (24 a (-1 1) 2)
      (24 a (-1 -1) 2))))
```

APPENDIX B

Questionnaire Used for PRL Study

This appendix contains the questionnaire used in the study (described earlier in this document) of how people understand programs.

DEBUGGING EXPERIMENT POST-MORTEM

We've taped your ramblings as you debugged the program. We've harassed you with questions as you tried to work. We've got the copy of the listing that you marked up. We've taken notes on what we saw you doing. Now...

These questions are to be answered immediately after you have completed the debugging task. Try to answer them as completely and accurately as possible. This is our last chance to figure out what you thought you were doing as you debugged the program.

1) What questions did you ask about the program's structure and design?

2) What sort of vocabulary did you use to refer to objects in the program, and the relations between them?

3) What sort of hypotheses did you construct, and how did you evaluate them?

4) What aids for searching through the program would you have liked?

5) Do you have any comments about the format of this experiment? Please vent your spleen here:

a] Suggest types of additional program documentation?
b] Would you like notes from author on program's intent?
c] Would labels that warn you about outdated code help?
d] Your gripe here...

6) Programmers often find themselves in the situation of having to maintain systems about which they know little. This experiment was an attempt to simulate that experience. We are in the process of defining a tool, called the PRL, to aid in program comprehension. We are soliciting your suggestions for such a tool, and your evaluation of our vision of the PRL.

Please look over the lists below. In it we have presented our breakdown of the classes of objects and relations you might want to talk about in analyzing a program. Would it be useful to be able to search for these types of objects and relations? How natural is the vocabulary? Feel free to suggest synonyms or rephrasings you find more natural. Also please add any useful concepts you think we have left out.

A) What types of objects would you like to be able to search for?

Text Strings

Syntactic Analysis
- Variable
- Function
- Let
- Loop
- Exits - "Show the exits from the splice-in loop of function FOO."

Cliches
- List-traversal
- Ordered-list
- Splice-in
- Priority-queue
- Enqueue-operation
- Dequeue-operation
- Hash-table
- Production-system
- Pattern / Trigger
- Action / Transformation

B) What types of relations are worth talking about?

Functional Composition
- Calls
- Called-by
- Recursive
- Mutually-recursive
- Main-loop
- Top-level-subroutine

Control Flow

(Sometimes/Always) Calls - "List the functions that function FOO always calls."

(Sometimes/Always) Returns

Data Flow

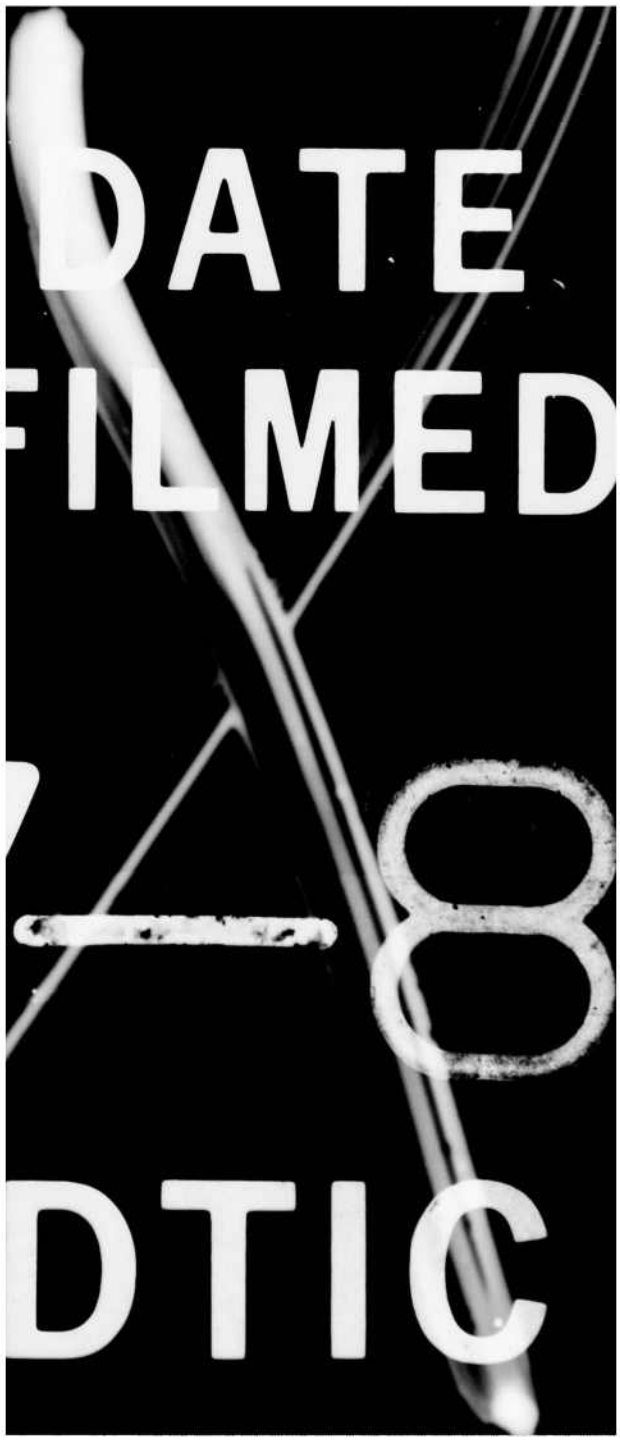
(Sometimes/Always) Accesses - "List the variables always accessed by function FOO."

(Sometimes/Always) Changes - "Find the variables sometimes changed by function FOO, and call it FOOLIST."

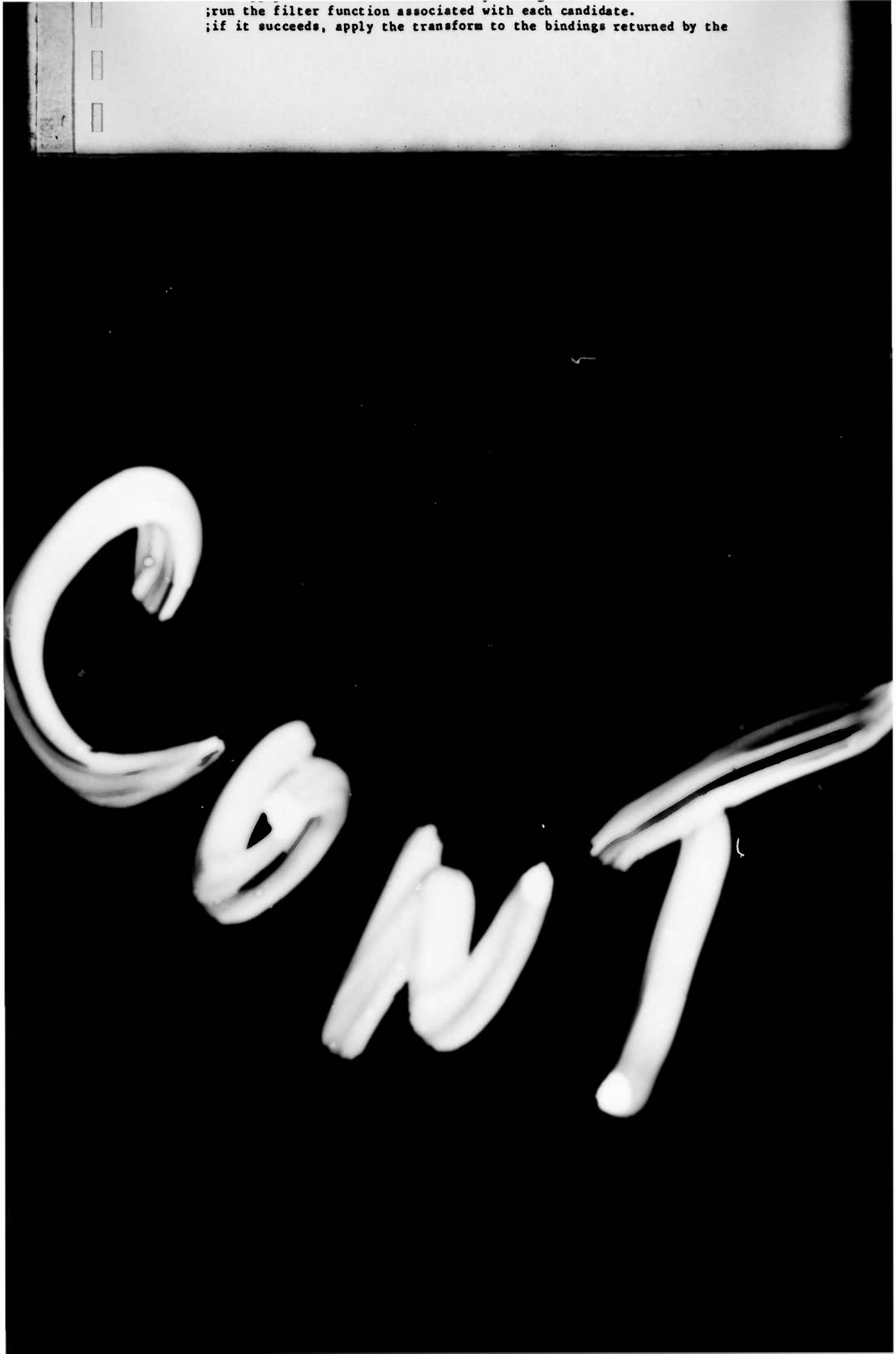
(Sometimes/Always) Side-effects

C) What forms of documentation would be especially helpful?

Main-routine
Data-structure
Input
Output
Side-effect
Precondition
Assumptions
Intentional-annotation = Collects segments of code that implement some particular purpose.
Hook = A comment describing why some code, not presently used, was designed in to facilitate some future expansion.
Inactive-code = A history of revisions.


DATE
FILMED
7-88
DTIC

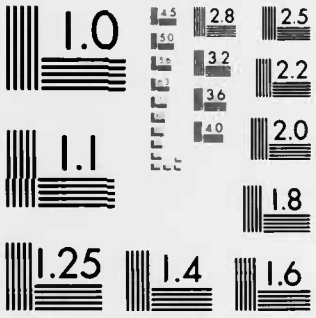
```
;run the filter function associated with each candidate.
;if it succeeds, apply the transform to the bindings returned by the
```



AD-A142 224 AN INFORMAL STUDY OF PROGRAM COMPREHENSION(U) ADVANCED INFORMATION AND DECISION SYSTEMS MOUNTAIN VIEW CA E A DOMESHEK ET AL. MAR 84 AI/DS-TM-1014-3

UNCLASSIFIED AFOSR-TR-84-0309 F49620-81-C-0067 F/G 12/1 NL

CONT



SUPPLEMENTARY

INFORMATION

Change

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A142 224

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): TM-1014-3	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 84-0309
6a. NAME OF PERFORMING ORGANIZATION: Advanced Information and Decision Systems; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Air Force Office of Scientific Research
6c. ADDRESS (City, State and ZIP Code): 201 San Antonio Circle, Suite 286, Mountain View CA 94040-1270	7b. ADDRESS (City, State and ZIP Code): Directorate of Mathematical & Information Sciences, Bolling AFB DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFOSR; 8b. OFFICE SYMBOL (If applicable): NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F49620-81-C-0067
8c. ADDRESS (City, State and ZIP Code): Bolling AFB DC 20332	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
61102F	2304	A7	

11. TITLE (Include Security Classification): AN INFORMAL STUDY OF PROGRAM COMPREHENSION,

12. PERSONAL AUTHOR(S): Brian P. McCune

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
~~Interim~~ Final	FROM 1/6/82 TO 31/5/83	MAR 1984	58

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

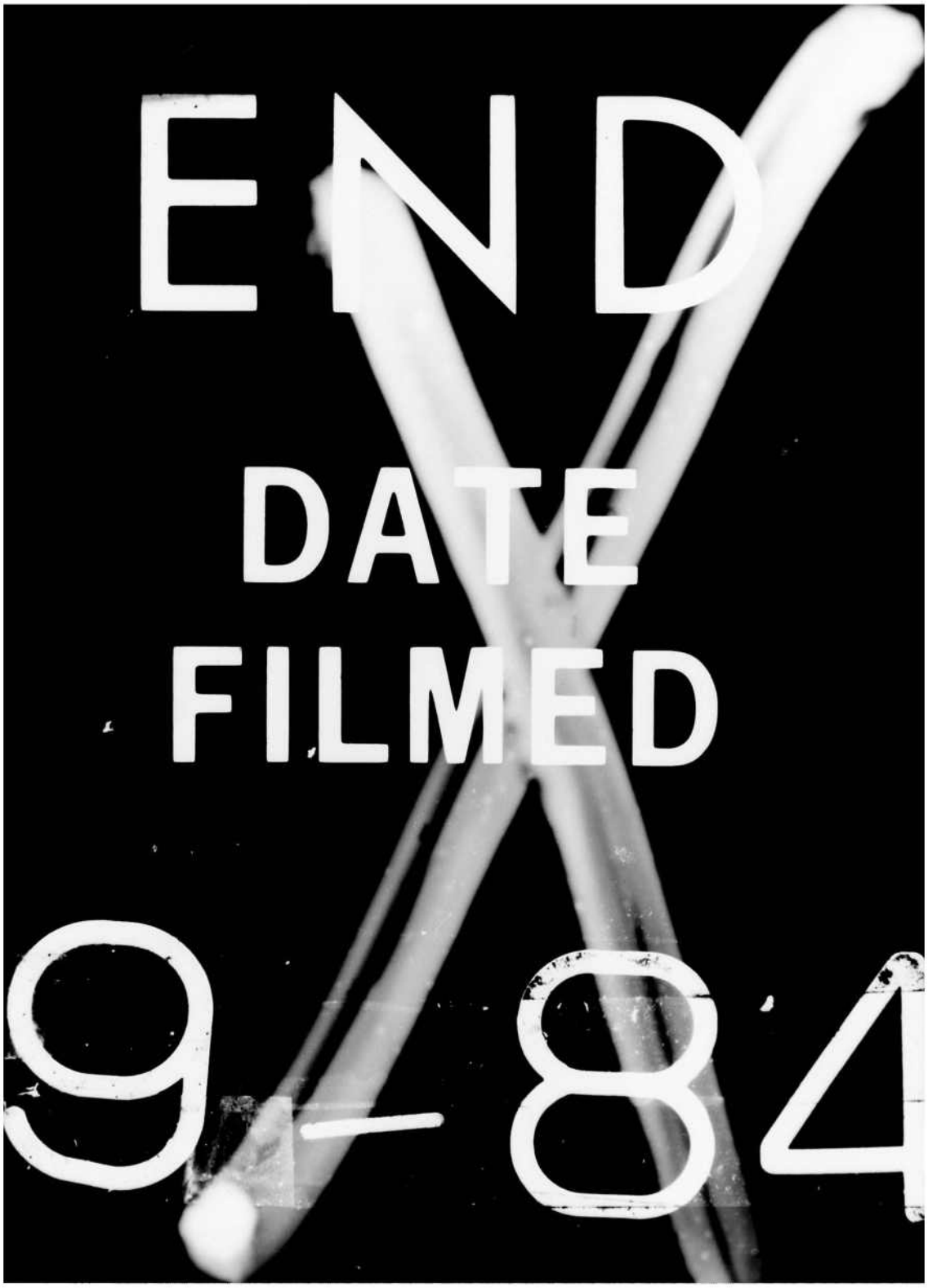
During this period, the four investigators produced 11 papers with titles including, "The intelligent program editor --- A knowledge based system for supporting program and documentation maintenance," "The multipurpose presentation system," "Rule based information retrieval," "Incremental informal program acquisition," and "Results in knowledge based program synthesis."

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Dr. Robert N. Buchal	(202) 767-4939	NM

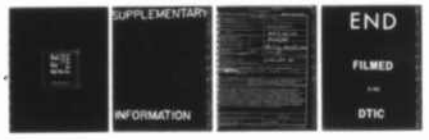
DD FORM 1473, 83 APR — EDITION OF 1 JAN 73 IS OBSOLETE

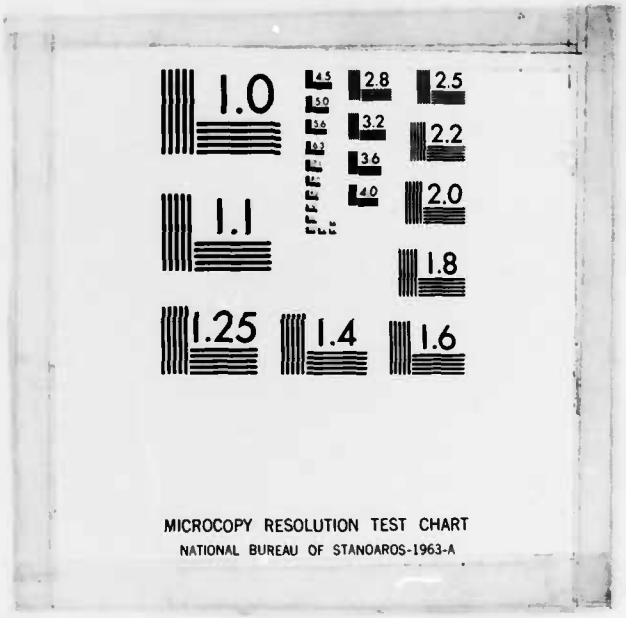
UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE


END
DATE
FILMED
9-84

AD-A142 224 AN INFORMAL STUDY OF PROGRAM COMPREHENSION(U) ADVANCED INFORMATION AND DECISION SYSTEMS MOUNTAIN VIEW CA E A DOMESHEK ET AL. MAR 84 AI/DS-TM-1014-3 3/3

UNCLASSIFIED AFOSR-TR-84-0309 F49620-81-C-0067 F/G 12/1 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SUPPLEMENTARY

INFORMATION

Change Bralla

ADA142224

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): TM-1014-3	5. MONITORING ORGANIZATION REPORT NUMBER(S): AFOSR-TR- 84-0309
6a. NAME OF PERFORMING ORGANIZATION: Advanced Information & Decision Systems; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: AFOSR/NM
6c. ADDRESS (City, State and ZIP Code): 201 San Antonio Circle, Suite 286, Mountain View, CA 94040	7b. ADDRESS (City, State and ZIP Code): Bolling AFB, DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: USAF, AFSC, Air Force Office of Scientific Research; 8b. OFFICE SYMBOL (If applicable): FQ8671	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: F49260-81-C-0007
8c. ADDRESS (City, State and ZIP Code): Building 410, Bolling AFB, D.C. 20332	10. SOURCE OF FUNDING NOS: PROGRAM ELEMENT NO 6/102F; PROJECT NO 2304; TASK NO A7; WORK UNIT NO.
11. TITLE (Include Security Classification): An Informal Study of Program Comprehension	
12. PERSONAL AUTHOR(S): Domeshek, Eric A.; Shapiro, Daniel G.; Dean, Jeffrey S.; McCune, Brian P.	
13a. TYPE OF REPORT: Technical Report Final / Annual; 13b. TIME COVERED: FROM 1 June 82 TO 31 May 83	14. DATE OF REPORT (Yr., Mo., Day): March 1984; 15. PAGE COUNT: 62
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB GR.	Program Reference Language (PRL), Extended Program Model (EPM), Intelligent Program Editor (IPE), program documentation, artificial intelligence (AI), knowledge base,

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes work performed during the second year of research on a Program Reference Language. During this year, a study was conducted in which protocols of programmers studying a new program (with the intent of debugging it) were analyzed, both for the vocabulary used and for indications of strategies adopted in their efforts at program comprehension. A sampling of programmers' natural vocabulary for referencing programs was gathered and analyzed. Preliminary steps were taken towards using this data as the basis for the design of a formal query language for the PRL. The study also raised some new issues bearing on the implementation of systems which use the PRL: individual differences imply the need for customization; context-sensitive information management is important; and useful user interface features were identified.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☒ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Robert N. Buchal	22b. TELEPHONE NUMBER (Include Area Code): (202) 767-4939; 22c. OFFICE SYMBOL

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE

END

FILMED

3-85

DTIC